OpenPort Protocol: A Security Governance Specification for AI Agent Tool Access

Genliang Zhu
Accentrust
Georgia Institute of Technology

Chu Wang
Accentrust
University of Illinois Urbana-Champaign

Ziyuan Wang
Duke University

Zhida Li
New York Institute of Technology-Vancouver

Qiang Li
Georgia Institute of Technology

Abstract

AI agents increasingly require direct, structured access to application data and actions, but production deployments still struggle to express and verify the governance properties that matter in practice: least-privilege authorization, controlled write execution, predictable failure handling, abuse resistance, and auditability. This paper introduces OpenPort Protocol (OPP), a governance-first specification for exposing application tools through a secure server-side gateway that is model- and runtime-neutral and can bind to existing tool ecosystems. OpenPort defines authorization-dependent discovery, stable response envelopes with machine-actionable `agent.*` reason codes, and an authorization model combining integration credentials, scoped permissions, and ABAC-style policy constraints. For write operations, OpenPort specifies a risk-gated lifecycle that defaults to draft creation and human review, supports time-bounded auto-execution under explicit policy, and enforces high-risk safeguards including preflight impact binding and idempotency. To address time-of-check/time-of-use drift in delayed approval flows, OpenPort also specifies an optional State Witness profile that revalidates execution-time preconditions and fails closed on state mismatch. Operationally, the protocol requires admission control (rate limits/quotas) with stable 429 semantics and structured audit events across allow/deny/fail paths so that client recovery and incident analysis are deterministic. We present a reference runtime and an executable governance toolchain (layered conformance profiles, negative security tests, fuzz/abuse regression, and release-gate scans) and evaluate the core profile at a pinned release tag using artifact-based, externally reproducible validation.

Index Terms

AI agents, tool exposure, authorization, least privilege, risk-gated writes, drafts and approvals, rate limiting, auditability, protocol specification.

I. Introduction

Modern AI agents operate by calling tools: structured functions or actions that read data and perform controlled writes in external systems. While early deployments rely on browser automation or one-off internal APIs, production environments require clear and verifiable guarantees: least-privilege access [1], tenant isolation, revocation, operational rate limits, and a complete audit trail.

OpenPort Protocol (OPP) targets the gap between "agents can call tools" and "tools can be safely exposed in real systems." The core observation is that tool discovery is not the hard part; authorization and governance are. A safe agent access layer must define the authorization surface, encode risk constraints for write operations, and make every decision auditable and operable. In other words, the "killer test" for production tool exposure is not a tool-list demo, but real-world authorization flows and predictable rate-limit behavior under failure and abuse.

Contributions. This paper makes three contributions:

1) A protocol-level specification for secure tool exposure to AI agents, including endpoints, response envelopes, tool metadata, and error taxonomy.
2) A security governance model that combines scopes and policy constraints with a draft-first write pipeline, high-risk safeguards (preflight, idempotency, and an optional state-witness profile for TOCTOU), mandatory audit events, and rate limits/quotas.
3) An engineering methodology for safe open-source evolution via conformance tests, fuzz/abuse regression, and release-gate checks that operationalize protocol invariants.

II. Problem Statement and Threat Model

OpenPort focuses on the security and governance gap between "tool exposure" and "safe tool exposure." While tool discovery and structured schemas are necessary for agents to interact with applications, they are not sufficient for production deployments.

Stewardship: Accentrust and the OpenPort Protocol authors.
Correspondence: research@accentrust.com.

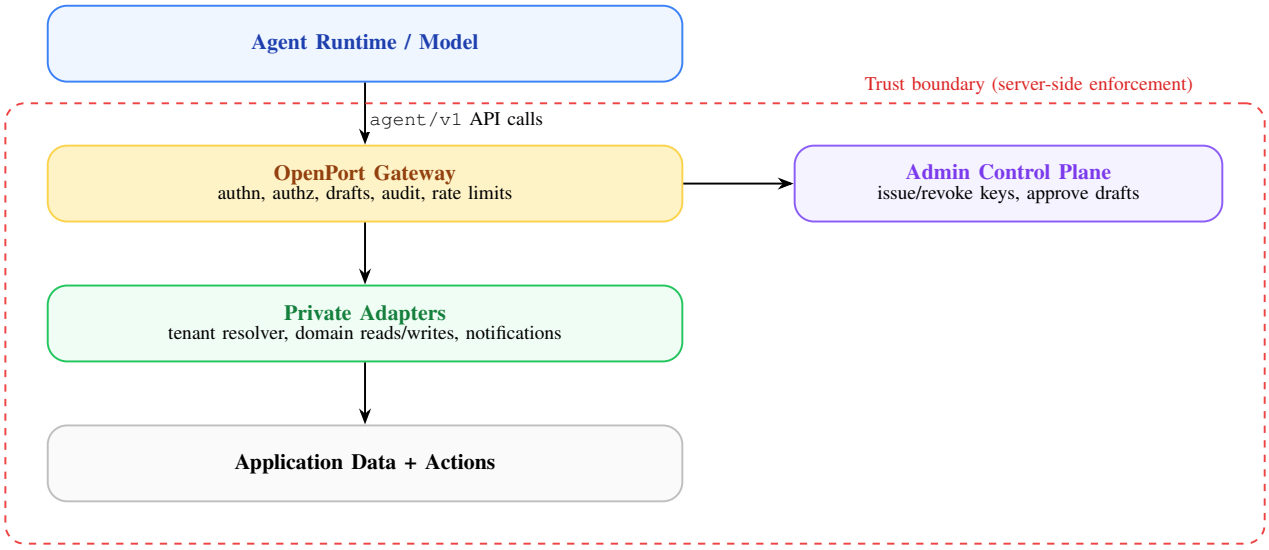

Fig. 1. System model and trust boundaries.

In practice, AI agents are probabilistic callers: they may call the wrong tool, call the right tool with the wrong arguments, retry unexpectedly, or be influenced by untrusted inputs (for example, prompt injection) to perform unsafe actions. Treating agent tool access as "just another API client" fails to capture these operational failure modes. This paper therefore frames the problem as a governance problem: how to expose application data and actions to agents through a narrow, explicitly authorized, rate-limited, and auditable interface that is safe under both malicious abuse and benign agent mistakes.

A. System Model and Trust Boundaries

OpenPort assumes a server-side gateway that mediates all agent access to an application. The gateway is the enforcement point responsible for authentication, authorization, rate limiting, audit emission, and write governance. Domain-specific logic and identity models remain behind adapters, allowing products to preserve internal schemas while still adopting a standardized governance surface. We assume the application operator controls the gateway and the admin control plane, and that all external access is over TLS.

B. Assets

OpenPort treats the following as protected assets:

- **Agent credentials**: agent keys and delegated tokens, including their metadata (key identifiers, issuance time, revocation state).
- **Tenant/workspace data**: the confidentiality of records reachable through read tools and the integrity of state modified by write tools.
- **Write intent and effects**: the correctness of drafts and executions, including protection against replay, duplicates, and unintended side effects.
- **Audit stream**: decision records and execution traces required for incident response and compliance.
- **Operational budget**: rate-limit and quota capacity, including protection against cost amplification (expensive queries, bulk exports, and high-frequency polling).

C. Assumptions

The threat model is intentionally pragmatic and matches typical production constraints:

- Transport is protected by TLS and the gateway is the single externally reachable enforcement point.
- Private adapters and application databases are not directly reachable by the agent runtime.
- Attackers may observe agent-visible content (tool descriptions, responses, and error messages) and may cause the agent to retry or to call unintended tools.
- Credentials may leak; therefore, OpenPort assumes that revocation, rate limiting, and auditability are required even for valid-looking requests.

D. Adversary Model

We consider both malicious and accidental failures that are characteristic of agent tool access:

- **External attacker**: attempts to steal tokens, enumerate tools, or exploit weak authorization checks.
- **Compromised agent runtime**: uses valid credentials to maximize impact (abuse, exfiltration).
- **Prompt injection / untrusted content**: influences an agent to request unsafe actions.
- **Benign failure**: retries, partial failures, and non-idempotent writes causing duplicate effects.
- **Operator mistakes**: overbroad scopes or misconfigured policies that silently increase blast radius.

We do not assume the model itself is trustworthy; instead, OpenPort constrains the *effects* of tool calls by requiring server-side enforcement and reviewable write paths.

E. Key Threats

When an application is opened to AI agents, a recurring set of production risks emerges:

1) **Token leakage** through prompts, logs, repositories, or CI systems.
2) **Tenant/workspace boundary bypass** due to client-supplied identifiers or inconsistent checks.
3) **Tool enumeration and capability leakage** where manifests, schemas, or error details reveal privileged capabilities or sensitive fields.
4) **Unapproved destructive writes** (delete/export/irreversible actions) caused by automation mistakes or prompt injection.
5) **Replay and duplicate effects** caused by retries, race conditions, or non-idempotent execution.
6) **Insufficient auditability** where decisions and side effects cannot be reconstructed.
7) **Abuse and overload** via excessive calls (DoS), expensive operations, or cost amplification.
8) **Sensitive data overexposure** through error details and logs that inadvertently capture payloads.

F. Security Goals

OpenPort Protocol addresses these threats by specifying server-side governance as protocol requirements. Rather than leaving security decisions as "implementation-defined," OpenPort promotes a small set of verifiable properties that can be tested in conformance suites and enforced in release gates. We state these properties as concrete security invariants:

1) **Deny-by-default authorization.** A request MUST be denied unless explicitly allowed by server-side scope grants and policy constraints; there is no implicit access.
2) **Server-enforced tenant/workspace boundaries.** Every read and write MUST be bound to a server-verified tenant/workspace context; client-supplied identifiers MUST NOT be trusted as the boundary.
3) **Authorization-dependent discovery.** `/manifest` MUST reflect current authorization: tools, fields, and capabilities that are not permitted MUST NOT be exposed via discovery responses.
4) **Immediate revocation.** Revoking a key or app MUST take effect immediately on all endpoints; denials due to revocation MUST be audited with a stable reason code.
5) **Draft-first writes.** Write actions MUST default to creating drafts; direct execution is only permitted when explicitly enabled by policy, and SHOULD be time-bounded.
6) **High-risk execution safeguards.** High-risk writes SHOULD require step-up confirmation and MUST support preflight impact hashing and idempotency keys to reduce unintended effects.
7) **Replay and duplicate protection.** For write intents, the pair {`appId`, `idempotencyKey`} MUST uniquely identify an execution outcome and MUST be safely de-duplicated.
8) **Operable abuse controls.** Implementations MUST enforce baseline rate limits and return stable 429 semantics; servers SHOULD include backoff guidance (for example, `Retry-After`) and SHOULD support quotas by tool and tenant.
9) **Audit-first governance.** Allow/deny/fail decisions and execution outcomes MUST emit structured audit events with stable reason codes and incident-response metadata (for example, key/app identifiers and request metadata).
10) **Data minimization.** Logs, error details, and audit payloads SHOULD redact sensitive fields and MUST NOT contain secrets such as bearer tokens.

G. Non-goals

OpenPort does not replace product identity systems, and it does not guarantee safety against a malicious administrator who controls the gateway and the admin plane. It also does not attempt to solve general prompt safety; instead, it constrains the *effects* of tool calls via governance requirements (least privilege, reviewable writes, and audited decisions).

TABLE I
THREATS AND REQUIRED CONTROLS IN OPENPORT.

Threat	Required protocol controls
Token leakage	Revocation, short-lived execution windows, rate limits, audit on every decision
Cross-tenant access	Server-side tenant resolver; deny-by-default; policy constraints; no client-trusted tenant IDs
Tool enumeration	AuthZ-dependent manifest; schema and error redaction; deny-by-default exposure
Destructive write misuse	Draft-first default; step-up confirmation; human approval chain; explicit risk tiers
Replay / duplicates	Idempotency keys; execution de-duplication per app and key
Audit gaps	Structured allow/deny/fail audit events with stable reason codes and request metadata
Abuse / DoS	Per-key/per-IP rate limiting; quotas by tool; predictable 429 behavior
Sensitive logging	Redaction rules; separation of operational logs vs immutable audit stream

III. PROTOCOL OVERVIEW

A. Design Principles

OpenPort is designed as a governance-first protocol surface: it standardizes the controls that must hold when exposing application data and actions to probabilistic callers (agent runtimes) rather than assuming a fully correct client. Its core principles are: (i) authorization-dependent, machine-readable tool discovery; (ii) server-side enforcement of authorization, policy constraints, and risk controls; (iii) stable, parseable response envelopes and `agent.*` reason codes so failures are operable; and (iv) a narrow write interface that defaults to reviewable drafts and can be strengthened with high-risk safeguards (preflight hashing, idempotency, and optional state-witness preconditions). Operationally, OpenPort treats admission control and auditability as first-class requirements: requests must be rate limited/quota controlled and every allow/deny/fail path must be reconstructable from structured audit events.

The protocol is domain-extensible: products implement domain reads and actions via adapters, while OpenPort fixes the governance semantics that must remain consistent across domains. Figure 2 summarizes this separation: OpenPort specifies governance semantics and a stable server-side interface, while compatibility with existing tool ecosystems is achieved through optional bindings rather than by delegating enforcement to each ecosystem. Bindings are informative and are evaluated separately from the core conformance claims in this paper.

B. Versioning and Compatibility

OpenPort uses explicit versioning in its URL path (for example, `agent/v1`) to support long-lived clients. Within a major version, servers SHOULD evolve the protocol in a backward-compatible manner (additive fields, additive tools, and stricter governance defaults that remain safe for existing clients). Clients SHOULD ignore unknown fields and treat tool discovery as dynamic, because permitted tools and schemas may change with scopes, policy constraints, and incident response actions such as revocation.

C. Core Objects

OpenPort standardizes a small set of objects used by both the protocol and governance layer:

- **Integration App**: a managed integration configuration with scopes and policy.
- **Agent Key**: a revocable token bound to an app (stored as a hash server-side).
- **Tool**: a machine-readable description of an action or read operation, including schemas and governance metadata.
- **Draft**: a reviewable representation of a requested write action.
- **Execution**: the outcome of executing a draft or an allowed write action.
- **Audit Event**: an immutable record of allow/deny/fail decisions and executions.

D. Tool Manifest

OpenPort treats tool discovery as an authorization-sensitive API. The agent runtime fetches `GET /api/agent/v1/manifest` to obtain the integration identity and a list of tools permitted under the presented credential. Each tool includes governance metadata (required scopes, risk tier, and confirmation requirements) and machine-readable schemas for inputs and outputs. The manifest MAY include HTTP hints (method and path) to support runtimes that want to provide clickable traces or debugging output, but correctness MUST NOT depend on clients calling arbitrary paths. Because the manifest is agent-visible, it MUST avoid capability leakage: tools and fields that are not authorized for the current integration MUST NOT appear, and error details SHOULD be redacted to avoid revealing privileged names.

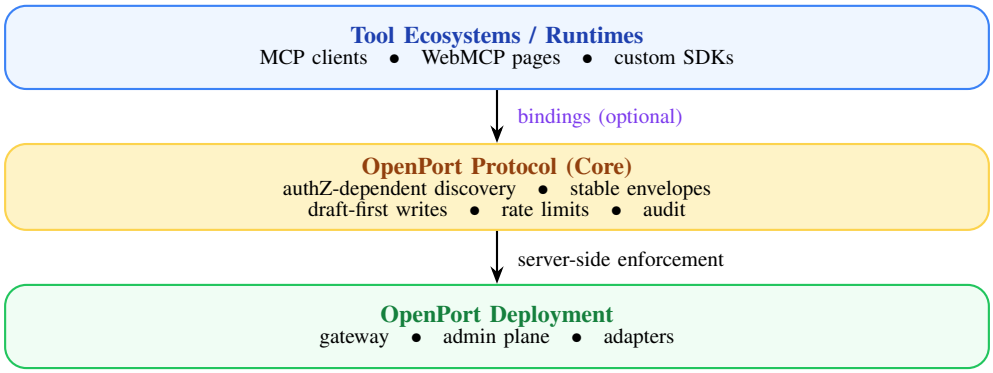

Fig. 2. OpenPort separates governance semantics from tool ecosystems: compatibility is achieved via optional bindings.

Formally, let $\mathcal{T}$ be the global set of tools, let $\mathrm{ReqScopes}(t)$ be the required scope set for tool t, and let $\mathrm{Scopes}(a)$ be the scopes granted to integration app a. Let $\mathrm{PolicyAllows}(a, t)$ denote a predicate that captures additional policy constraints on exposure (for example, field redaction and disabled capabilities). OpenPort defines the visible tool set as:

$$\mathrm{VisibleTools}(a) = \{t \in \mathcal{T} \mid \mathrm{ReqScopes}(t) \subseteq \mathrm{Scopes}(a) \\ \wedge \ \mathrm{PolicyAllows}(a, t)\}. \tag{1}$$

This equation expresses the core "authorization-dependent discovery" requirement: discovery output is a function of current authorization state, not a static listing.

OpenPort uses a small, opinionated risk scale to support consistent governance. **Low** risk tools are typically read-only or trivially reversible operations; **medium** risk tools perform write operations that are expected to be reversible or bounded; and **high** risk tools include destructive writes, bulk exports, or other actions with significant blast radius. Risk tier is part of tool metadata to support both server-side enforcement (draft-first defaults and high-risk safeguards) and client-side UX (when to request preflight, how to explain required approvals).

```
{
  "name": "transaction.hard_delete",
  "description": "Permanently delete a transaction.",
  "requiredScopes": ["transaction.delete"],
  "risk": "high",
  "requiresConfirmation": true,
  "http": { "method": "POST", "path": "/api/agent/v1/actions" },
  "inputSchema": { "type": "object", "properties": { "payload": { "type": "object" } } },
  "outputSchema": { "type": "object", "properties": { "deleted": { "type": "object" } } }
}
```

Listing 1. Conceptual tool object in OpenPort manifest.

E. Core Endpoints

OpenPort defines a minimal agent-facing protocol surface:

```
GET  /api/agent/v1/manifest
GET  /api/agent/v1/ledgers
GET  /api/agent/v1/transactions
POST /api/agent/v1/preflight
POST /api/agent/v1/actions
GET  /api/agent/v1/drafts/{id}
```

Listing 2. OpenPort Protocol agent/v1 endpoints (conceptual).

Read endpoints (for example, `/ledgers`, `/transactions`) are illustrative; implementations MAY expose additional domain reads as long as they follow the stable envelope, enforce the same authorization model, and emit audit events. In contrast, writes are intentionally funneled through a small number of governance-aware endpoints.

An admin control plane is required for safe operation and is described below.

F. Domain Extensibility and Adapters

OpenPort intentionally separates *governance semantics* from *domain semantics*. Domain adapters implement product-specific reads and actions (for example, accounting, CRM, ticketing) and translate them into the OpenPort tool model. This boundary has two protocol consequences:

- **Extensible reads.** Implementations MAY add new read endpoints, but each MUST be declared (directly or indirectly) through authorization-dependent discovery and MUST enforce the same scope/policy/tenant controls as core endpoints.
- **Centralized writes.** Implementations SHOULD route writes through a small number of governance-aware endpoints (for example, `/preflight` and `/actions`) so that draft-first, step-up, idempotency, and audit behavior remains consistent across all write tools.

In the reference runtime, this takes the form of a tool registry that binds each tool name to an adapter-backed function and attaches governance metadata (required scopes, risk tier, and optional impact computation).

G. Admin Control Plane (Required)

OpenPort relies on a human-managed admin control plane to close the authorization loop. While product identity is out of scope, production deployments require a minimal set of governance actions:

- create and revoke integration apps and keys (issuance and rotation)
- update policy constraints and auto-execute windows
- list, approve, and reject drafts
- export audit events and operational summaries

This admin plane is also the natural place to enforce organizational procedures (for example, separation of duties) and to integrate with incident response workflows.

H. Write Lifecycle: Preflight, Drafts, Execution

OpenPort separates three phases of write operations:

- **Preflight** (`POST /preflight`) optionally computes an impact summary and returns an impact hash that can be bound to later execution. Under an optional stronger governance profile, preflight MAY also return a `stateWitnessHash` that binds execution to a server-observed resource version/state. Implementations MAY also return a short-lived preflight handle (`preflightId`) that allows clients to reuse the exact payload and hash without resending or regenerating content. High-risk policies may require preflight.
- **Request** (`POST /actions`) submits a tool name and payload, and produces either a draft (default) or an execution result (when explicitly permitted).
- **Review/Execute** occurs via the admin plane for draft approval, followed by execution under policy constraints (idempotency, preflight hash binding, and step-up requirements).

Drafts expose a stable status model (for example, draft, confirmed, canceled, failed) and are queryable via `GET /drafts/{id}`. This allows agent runtimes to treat writes as asynchronous and reviewable by design.

1) Preflight Hash Binding: Preflight binds execution to a computed impact summary by returning a stable digest over canonical inputs. Let $H(\cdot)$ be a cryptographic hash function and let $\mathcal{C}(\cdot)$ be a deterministic canonical encoding (for the reference profile, RFC 8785 JSON Canonicalization Scheme (JCS) over JSON values [2]). Let $I(t, x)$ be the server-side impact function for tool t and payload x (possibly adapter-backed). OpenPort defines the preflight hash as:

$$h = H\big(\mathcal{C}(t, x, I(t, x))\big). \tag{2}$$

If a policy requires preflight for a request, the server MUST deny execution unless the client supplies a preflight hash that matches the server-computed value in Eq. (2). To make this binding robust under agent non-determinism, clients SHOULD treat `/preflight` and `/actions` as one write intent: do not generate a second payload after preflight. A server MAY provide `preflightId` as a convenience handle that resolves to the cached payload and hash. Such handles SHOULD be TTL-bound and scoped to the credential context (for example, app/key/actor) that created them. If a `preflightId` is provided but cannot be resolved (expired, unknown, or wrong credential context), the server SHOULD fail closed with agent.preflight_not_found.

2) State Witness Preconditions (Optional Strong Profile): Preflight hash binding protects against payload non-determinism and intent swapping, but it does not fully address time-of-check to time-of-use (TOCTOU) risks when human approvals are delayed and the underlying world state can change. OpenPort therefore defines an optional stronger governance profile: *State Witness / Preconditions*. Implementations MAY support this profile. Implementations that claim this profile MUST enforce the preconditions described below when a request or draft is bound to a state witness hash.

Let $W(t, x)$ be a server-side witness function that returns a minimal, non-secret snapshot of the relevant resource state for tool t and payload x (for example, an HTTP `ETag`-like value, a `resourceVersion`-like field, or an adapter-defined version tuple). OpenPort defines a witness hash:

$$w = H\big(\mathcal{C}(W(t, x))\big). \tag{3}$$

If a draft is bound to w (or a client supplies `stateWitnessHash` directly), the server MUST recompute the current witness at execution time and MUST fail closed unless the current witness hash matches w. On mismatch, the server MUST deny with a stable reason code agent.precondition_failed and MUST NOT execute side effects. Clients SHOULD treat this as a state change: rerun `/preflight`, refresh operator approval, and avoid blind retries. This mechanism is specified as an optional profile extension and is evaluated separately from the pinned `v0.1.0` core evaluation claims in Section XI.

3) Action Request Schema: Action requests use an explicit tool name and a free-form payload whose structure is governed by the tool's input schema in the manifest. To support safe retries and operational tracing, OpenPort defines optional fields for request correlation and replay protection:

```
{
  "action": "transaction.hard_delete",
  "payload": { "...": "..." },
  "preflightId": "pfl_...",
  "execute": true,
  "forceDraft": false,
  "requestId": "req_...",
  "idempotencyKey": "idem_...",
  "justification": "operator intent for high-risk execution",
  "preflightHash": "sha256(...preflight impact...)",
  "stateWitnessHash": "sha256(...witness...)"
}
```

Listing 3. Conceptual OpenPort action request.

When `preflightId` is provided, servers MAY reuse a cached payload and preflight hash, allowing clients to omit `payload` as long as the resolved payload matches the requested action and credential scope. The server MUST decide whether execution is permitted. If execution is not permitted, the server MUST return a draft and MUST provide a machine-readable denial reason code for the auto-execute request. If an idempotency key is provided and a matching execution already exists, the server SHOULD return the prior execution outcome rather than re-executing the tool, to make agent retries safe.

4) Auto-Execute Eligibility Predicate: OpenPort treats auto-execution as an explicit, time-bounded capability. Let a be an integration app, let t be a tool, and let $\mathrm{cfg}(a)$ be the auto-execute configuration state. Let $\exp(a, t)$ denote the configured expiration timestamp for the relevant auto-execute window and let $\mathrm{AllowList}(a, t)$ denote an optional allowlist of tool names. Let $\mathrm{Req}(r)$ be the action request fields (execute, forceDraft, justification, idempotencyKey, and either `preflightHash` or a `preflightId` that resolves to one, and optionally `stateWitnessHash` under the State Witness profile). OpenPort defines an eligibility predicate:

$$\begin{aligned}
\mathrm{AutoExecAllowed}(r) = \mathrm{Req}(r).\mathrm{execute} \wedge \neg \mathrm{Req}(r).\mathrm{forceDraft} \\
\wedge\ \mathrm{Enabled}(\mathrm{cfg}(a), t) \wedge (\mathrm{ts} < \exp(a, t)) \\
\wedge\ \Big(\mathrm{AllowList}(a, t) = \emptyset \vee \\
\quad t \in \mathrm{AllowList}(a, t)\Big) \\
\wedge\ \Big(\mathrm{risk}(t) \neq \texttt{high} \vee \\
\quad \big(\mathrm{Req}(r).\mathrm{justification} \neq \emptyset \wedge \\
\quad \mathrm{HighRiskGuards}(r)\big)\Big),
\end{aligned} \tag{4}$$

where $\mathrm{HighRiskGuards}(r)$ includes idempotency and preflight requirements when configured, and preflight hash matching uses Eq. (2). If $\mathrm{AutoExecAllowed}(r)$ is false, the server MUST return a draft and MUST return a stable denial reason code.

5) Idempotency as a Deterministic Execution Map: Idempotency is expressed as a deterministic mapping from an integration app and idempotency key to an execution outcome. Let a be an integration app identifier and let k be an idempotency key. OpenPort defines:

$$E(a, k) \to \texttt{Execution}, \tag{5}$$

with the invariant that if $E(a, k)$ is already defined, then subsequent requests carrying the same (a, k) MUST NOT re-execute side effects and SHOULD return the existing execution record. This invariant converts agent retries and partial failures into safe, replayable outcomes.

6) Policy Snapshotting: To make approvals and incident response tractable, OpenPort recommends snapshotting the governance-relevant policy inputs at the time a draft is created (for example, required scopes, risk tier, and auto-execute configuration). This supports later review of "what the system believed was permitted" at the time of the request, even if policies change before approval.

I. Response Envelope and Error Taxonomy

Every endpoint returns a stable response envelope:

- success: { `ok: true, code, data` }
- error: { `ok: false, code, message, details?` }

This stability is a core requirement for agent runtimes that need predictable parsing and recovery strategies. The `code` field is a stable machine-facing identifier. For operational controls such as rate limiting, servers SHOULD return 429 with backoff guidance (for example, `Retry-After`) [3], [4]. For authorization denials, servers SHOULD return stable denial codes to support safe retry and operator escalation. OpenPort uses a small error taxonomy to make denial behavior predictable (invalid/expired token, scope or policy denial, unknown/invalid action, preflight required/mismatch/not found, idempotency required/replay, and rate limited). Optional stronger profiles MAY add additional stable denial codes; for example, the State Witness profile introduces agent.precondition_failed when an execute-time precondition does not hold.

IV. Authorization Model

A. Token Types

OpenPort supports two authorization modes that cover most production deployments:

1) **Integration tokens** for machine-to-machine access, bound to an integration app/key and constrained by server-side scopes and policy constraints.
2) **Delegated tokens** for "act on behalf of" flows (for example, OAuth 2.0), which MUST be mapped to minimal OpenPort scopes and policy windows before any tool access is granted.

The protocol does not mandate a specific identity provider; it mandates that authorization decisions are made server-side and are auditable.

B. Credential Presentation and Validation

Agent requests present credentials as bearer tokens (for example, `Authorization: Bearer <token>`). To minimize leakage impact, OpenPort recommends opaque (reference) tokens that are validated by server-side lookup; if self-contained tokens are used, they SHOULD be short-lived and MUST support immediate revocation.

Validation MUST check, at minimum:

- token presence and parseability
- token revocation and expiration
- integration app status (active vs revoked/disabled)
- network policy constraints (for example, IP allowlists) when configured
- baseline abuse controls (rate limits) to bound impact under leakage

Failures MUST return stable denial codes (for example, invalid token, expired token, policy denied, rate limited) so agent runtimes can implement safe retry and operator escalation.

C. Authorization Decision Algorithm

OpenPort constrains authorization to a deterministic decision function. For each request, servers SHOULD evaluate in a fixed order:

1) **Authenticate**: validate the credential and resolve the integration app.
2) **Apply network policy**: enforce IP allowlists or other network constraints.
3) **Rate limit**: apply per-key/per-IP limits to bound abuse.
4) **Authorize by scope**: ensure the tool's required scopes are granted to the app.
5) **Authorize by policy**: apply ABAC-style constraints to restrict the data domain and output surface.
6) **Enforce tenant/workspace boundary**: verify all domain access is within the app's permitted boundary.
7) **Audit**: emit allow/deny/fail events for authenticated requests with stable reason codes and incident-response metadata.

This ordering is not cosmetic: it is what makes decisions predictable, testable, and operable.

1) Formal Authorization Predicate: Let a request be $r = (\tau, t, x, \mathrm{ip}, \mathrm{ts})$ where τ is a credential, t is a tool name, x is the payload, and ip and ts are request metadata. Let $\mathrm{App}(\tau)$ resolve the integration app and key for a valid credential. OpenPort defines authorization as the conjunction of server-side predicates:

$$\begin{aligned}\mathrm{Allow}(r) = \mathrm{Authn}(\tau) \ \wedge \ \mathrm{Net}(\mathrm{ip}, \mathrm{App}(\tau)) \ \wedge \ \mathrm{RL}(r) \\ \wedge \ \mathrm{ScopeOK}(t, \mathrm{App}(\tau)) \ \wedge \ \mathrm{PolicyOK}(t, x, \mathrm{App}(\tau)) \\ \wedge \ \mathrm{BoundaryOK}(t, x, \mathrm{App}(\tau)).\end{aligned} \tag{6}$$

To ensure stable reason codes, implementations can map the *first failing predicate* in the evaluation order to a denial code. If the ordered predicates are $(p_1, \ldots, p_n)$, the denial index is:

$$i^* = \min\{i \mid p_i(r) = \texttt{false}\}, \tag{7}$$

and the reason code is a deterministic function $\mathrm{Code}(p_{i^*})$.

D. Least Privilege: Scopes

Scopes are static capability labels required by tools (for example, `transaction.read`, `transaction.write`). Tools declare `requiredScopes` in the manifest. The server MUST enforce deny-by-default scope checks; absence of a required scope implies denial. When multiple scopes are required, the semantics are conjunctive (all required scopes must be present).

E. Least Privilege: Policy Constraints (ABAC)

Policy constraints restrict the *data domain* and *presentation surface* beyond what scopes can express. OpenPort models policy as ABAC-style constraints [5] and treats it as mandatory enforcement state. Typical constraints include:

- **Network policy**: IP allowlists for machine clients.
- **Data policy**: allowed resource sets (for example, allowed ledger IDs or organization IDs), bounded time windows for queries, and field-level redaction toggles for sensitive attributes.

Policy MUST be enforced server-side and SHOULD be reflected in discovery responses (for example, hiding tools or fields that would be denied).

1) A Bounded Query Window Constraint: A common production control is bounding query time windows to limit both data exposure and cost amplification. Let $d_{\max}$ be the maximum number of days allowed by policy, and let (s, e) be the effective start and end dates for a request after defaulting missing bounds. OpenPort requires the constraint:

$$\Delta(s, e) = \frac{e - s}{\mathrm{day}} \leq d_{\max}, \tag{8}$$

otherwise the server MUST deny with a policy reason code.

2) Policy Effects on Output: Policy constraints may also restrict the *presentation surface* of returned data. For example, a policy can disable access to sensitive fields; servers SHOULD implement deterministic redaction so that clients can treat outputs as stable and safe to display. When redaction occurs, servers SHOULD disclose which fields were redacted using non-sensitive identifiers (for example, field paths) and SHOULD record redaction outcomes in audit events without logging raw sensitive values. This behavior can be formalized as a function that returns both a presented object and a set of redacted field paths:

$$\mathrm{Present}(o, p) \to (o', F), \tag{9}$$

where o is the raw domain object, p is the effective policy, o' is the redacted presentation, and F is the set of redacted fields.

F. Tenant and Workspace Boundary Enforcement

Tenant isolation is a protocol requirement, not an implementation detail. In multi-tenant deployments, an integration app may be scoped to a specific workspace/organization. Servers MUST enforce this boundary using server-side resolution (for example, by fetching resource metadata through adapters) and MUST NOT trust client-provided tenant identifiers as the source of truth. When policies restrict allowed organizations or resource IDs, those checks MUST compose with workspace boundaries.

G. Reason Codes and Error Semantics

OpenPort requires stable reason codes for denials so that agent runtimes can implement safe behavior (refresh discovery, request approval, back off, or escalate to an operator). Reason codes SHOULD be scoped and human-interpretable (for example, `agent.token_invalid`, `agent.scope_denied`, `agent.policy_denied`, `agent.rate_limited`), and MUST avoid embedding secrets or internal identifiers. OpenPort standardizes a minimum set of `agent.*` codes for interoperability; implementations MAY extend with additional codes as long as existing semantics remain stable. For requests that fail before an integration app can be resolved (for example, missing/invalid token), implementations MAY prefer aggregate security metrics over per-request audit events to avoid log amplification; for authenticated requests, auditability is mandatory.

```
Admin  -> Admin API: POST /api/agent-admin/v1/apps (scopes, policy)
Admin  -> Admin API: POST /api/agent-admin/v1/apps/{id}/keys (rotate/issue)
Agent  -> Agent API : GET /api/agent/v1/manifest (Bearer key)
Server -> Audit     : allow/deny + code + keyId/appId
```

Fig. 3. Integration token provisioning and authorized tool discovery.

```
Agent  -> Agent API : GET /api/agent/v1/ledgers
Server -> Audit     : allow/deny + code + resultCount
Agent  -> Agent API : GET /api/agent/v1/transactions?ledgerId=...
Server -> Audit     : allow/deny + code + redactedFields
```

Fig. 4. Authorized reads with policy enforcement and audit emission.

H. Lifecycle: Issuance, Rotation, Revocation

Production authorization requires full lifecycle support: key issuance, parallel keys for rotation, and revocation that is effective immediately. Credentials SHOULD support metadata useful for operations (token prefix, last-used timestamps, expiration) without exposing secrets.

1) Issuance: An integration app is provisioned with an explicit scope set and policy constraints. Keys are issued to apps and SHOULD be displayed only once at issuance; servers SHOULD store only a non-reversible form of the secret (for example, a hash).

2) Rotation: Rotation SHOULD allow parallel validity windows (old and new keys) to avoid downtime and to support staged rollout. Servers SHOULD provide stable key identifiers (not secrets) so operators can audit which credentials were used.

3) Revocation and Emergency Disablement: Revocation MUST take effect immediately across all endpoints, including discovery. OpenPort treats revocation as a first-class security control: denials due to revocation MUST generate audit events with stable reason codes. Implementations SHOULD support an emergency disable switch at the app level to stop all agent access quickly during an incident response.

V. Authorization Flows

This section specifies the expected end-to-end authorization flows that make OpenPort deployable in production. The goal is not to prescribe an identity system, but to define an authorization *closed loop*: issuance, use, decision logging, and revocation.

A. Provisioning, Issuance, and Tool Discovery (Integration Tokens)

An operator provisions an integration app with explicit scopes and policy constraints, then issues an agent key. The agent runtime uses the key to discover permitted tools via the manifest. Discovery is part of the authorization surface: the manifest MUST be authorization-dependent (Eq. (1)) and MUST not leak tools or sensitive fields that are not permitted for the current integration.

B. Authorized Reads and Policy Effects

After discovery, the agent runtime calls read endpoints to retrieve data. Read access is governed by the same decision algorithm (Eq. (6)) and MUST enforce tenant/workspace boundaries and ABAC constraints (resource allowlists, bounded query windows, and redaction). When policies disable sensitive fields, servers SHOULD redact deterministically and SHOULD disclose redacted field paths as in Eq. (9).

C. Request Evaluation and Reason Codes

For each request, OpenPort requires server-side evaluation in a fixed order: authentication (token validity), scope matching, policy constraints (ABAC), and tenant/workspace boundary enforcement. Each denial SHOULD map to a stable reason code (for example, `agent.token_invalid`, `agent.token_expired`, `agent.scope_denied`, `agent.policy_denied`, `agent.forbidden`, `agent.rate_limited`). These codes enable safe agent retry behavior, operator debugging, and security analytics without exposing sensitive internal details.

D. Write Requests: Draft-First and Optional Auto-Execute

Write requests are submitted via `POST /api/agent/v1/actions`. OpenPort requires draft-first behavior: in the absence of explicit, time-bounded policy enablement, write requests MUST produce drafts rather than execute side effects. Auto-execute eligibility is governed by Eq. (4) and is designed to be auditable and safely reversible through revocation.

For high-risk tools, the agent runtime SHOULD call `POST /api/agent/v1/preflight` to obtain an impact summary and hash (Eq. (2)) before requesting execution. If auto-execution is denied (disabled, expired, allowlist mismatch, missing

```
Agent  -> Agent API : POST /api/agent/v1/preflight (action, payload)
Server -> Agent     : impact, impactHash, preflightId, stateWitnessHash?
Agent  -> Agent API : POST /api/agent/v1/actions (execute?, idem?, preflightHash?, preflightId?,
    stateWitnessHash?)
Server -> Agent     : Draft (default) or Execution (if allowed)
Server -> Audit     : allow/deny + code + draftId/executionId
```

Fig. 5. Write request flow with optional preflight and draft-first defaults.

```
Admin -> Admin API: POST /api/agent-admin/v1/apps/{id}/keys (issue new key)
Agent -> Agent API : use new key (progressive rollout)
Admin -> Admin API: POST /api/agent-admin/v1/keys/{id}/revoke (revoke old key)
Agent -> Agent API : old key fails immediately (agent.token_invalid)
```

Fig. 6. Key rotation and immediate revocation.

justification, missing idempotency, or preflight mismatch), the server MUST return a draft and a machine-readable denial code so operators can decide whether to approve. If an idempotency key is present and a prior execution exists for the same $(\texttt{appId}, \texttt{idempotencyKey})$, the server SHOULD return the prior execution outcome (Eq. (5)) to make retries safe.

E. Admin Review and Execution

Drafts provide a reviewable representation of write intent. The admin control plane lists drafts, inspects context (including policy snapshots), and approves or rejects requests. Approval triggers execution under the recorded policy constraints; rejection cancels the draft. This human-in-the-loop step is a governance requirement for high-risk operations and is the primary mechanism to mitigate prompt-injection-driven misuse.

OpenPort supports asynchronous client behavior by allowing agents to poll draft status:

```
GET  /api/agent/v1/drafts/{id}
GET  /api/agent-admin/v1/drafts?status=draft
POST /api/agent-admin/v1/drafts/{id}/approve
POST /api/agent-admin/v1/drafts/{id}/reject
```

Listing 4. Draft review and polling endpoints (conceptual).

Clients SHOULD implement backoff for polling and treat drafts as non-final until an execution outcome is recorded.

F. Rotation, Revocation, and Incident Response

Authorization safety requires that credentials can be rotated and revoked without downtime. Rotation issues a new key while keeping the old key valid for a controlled overlap window; revocation invalidates a key or an entire app immediately across all endpoints, including `/manifest`. Implementations SHOULD support emergency disablement at the app level to stop all agent traffic quickly during an incident.

G. Client Recovery Guidance

Stable reason codes make agent runtimes safer by enabling deterministic recovery logic. Table II summarizes recommended responses for common denial codes. These recommendations are intentionally aligned with the conformance invariants in Table V: a server that satisfies the invariants should enable clients to implement Table II without relying on model-specific heuristics.

H. Delegated Authorization Binding (OAuth 2.0)

OpenPort is compatible with delegated "act on behalf of" use cases, commonly implemented via OAuth 2.0 [6]. However, delegated identity is insufficient without governance:

- delegated tokens MUST map to minimal OpenPort scopes;
- delegated sessions MUST apply policy windows (time bounds, resource sets, field redaction);
- revocation MUST be effective immediately (including user consent withdrawal).

The binding MAY use token introspection or local validation, but the resulting authorization decision MUST be enforced server-side and MUST be auditable.

TABLE II
OPENPORT REASON CODES AND RECOMMENDED CLIENT BEHAVIOR (BASELINE SET WITH OPTIONAL PROFILE EXTENSIONS).

Reason code	**Recommended client action**
agent.token_invalid agent.token_expired	Stop; request a new key/token; refresh discovery after re-authentication.
agent.scope_denied	Refresh `/manifest`; request additional scopes only if explicitly approved.
agent.policy_denied agent.forbidden	Surface to operator; adjust query/window/resource selection; do not retry blindly.
agent.action_unknown agent.action_invalid	Do not retry blindly; refresh `/manifest` and validate tool name and input schema.
agent.preflight_required agent.preflight_mismatch agent.preflight_not_found	Run `/preflight` and retry with returned hash (or a valid `preflightId`); treat mismatch as a change in impact and do not execute blindly.
agent.precondition_failed *(State Witness profile)*	Treat as state change (TOCTOU); rerun `/preflight` to obtain a fresh witness/hash and require renewed operator approval before execution.
agent.idempotency_required	Generate a stable idempotency key per write intent and retry once.
agent.idempotency_replay	Treat as success; reuse returned execution outcome; do not re-execute.
agent.auto_execute_disabled agent.auto_execute_expired agent.auto_execute_denied	Accept draft result; route to admin approval rather than retrying execution.
agent.draft_not_found agent.draft_already_final	Stop; refresh draft status from the gateway; treat as non-retriable and require operator review.
agent.step_up_required agent.step_up_invalid	Complete step-up verification through the operator channel and retry within the allowed window.
agent.rate_limited	Back off; respect `Retry-After` if present; reduce polling; consider caching `/manifest`.

```
User  -> IdP       : OAuth authorize (consent)
Agent -> IdP       : Obtain delegated access token
Agent -> Agent API : Call tool (delegated token)
Server-> AuthZ     : Map claims -> scopes + policy window
Server-> Audit     : allow/deny + reason + subject + appId
```

Fig. 7. Delegated authorization binding: identity plus governance.

VI. RISK-GATED WRITES

Write operations are the primary risk surface for agent tooling because they create side effects and are harder to contain than reads. OpenPort therefore treats "write access" as a governed capability rather than a default API behavior. The protocol's baseline write posture is **draft-first**: write requests create reviewable drafts unless auto-execution is explicitly enabled and time-bounded.

A. Draft-First Semantics

Write requests are submitted via `POST /api/agent/v1/actions` using a tool name and a payload (schemas are discovered via the manifest). The server MUST evaluate authorization and policy constraints server-side (Eq. (6)) before any side effects can occur. Authorization failures (invalid token, missing scope, policy denial) remain hard errors with stable reason codes. If the request is authorized but not eligible for immediate execution, the server MUST return a draft. If execution was explicitly requested (for example, `execute=true`) but denied by auto-execute policy, the server MUST attach a stable machine-readable denial reason code for the *auto-execute request*. Clients MAY explicitly force drafting (for example, `forceDraft=true`) to require operator review even when an auto-execute window is enabled. This "fail closed into draft" rule converts many unsafe behaviors (tool misuse, prompt injection, retries) into a review queue rather than side effects.

B. Draft and Execution Objects

OpenPort separates *intent* (draft) from *effect* (execution). A draft represents a write request that can be reviewed and approved by an operator. An execution represents the outcome of running a tool under enforced constraints.

At minimum, a draft record SHOULD include:

- identifiers: `draftId`, `appId`, `keyId`, actor identity;
- intent: tool name (`actionType`) and payload;
- governance: risk tier, `auto_execute_requested`, optional `justification`;

```
Agent  -> Agent API: POST /api/agent/v1/actions (tool, args, execute?)
Server -> Audit    : allow/deny + code + risk tier + draftId
Server -> Agent    : Draft (default) or Execution (if allowed)
Admin  -> Admin API: POST /api/agent-admin/v1/drafts/{id}/approve
Server -> Target   : Execute (idempotency + optional preflight hash)
Server -> Audit    : execution success/fail + executionId
```

Fig. 8. Draft-first write pipeline with approval and high-risk safeguards.

TABLE III
DRAFT LIFECYCLE STATES AND TRANSITIONS (REFERENCE PROFILE).

Status	Meaning	Transitions (trigger)
`draft`	Pending operator decision; no side effects have been executed.	`confirmed` (approve or eligible auto-execute), `canceled` (reject).
`confirmed`	Approved for execution; a successful execution outcome should be recorded for completion.	`failed` (execution attempt fails).
`canceled`	Rejected by operator; terminal.	none.
`failed`	Execution attempted and failed; terminal.	none.

- safeguards: optional `preflight` impact summary and `preflightHash`, optional `idempotencyKey`;
- policy snapshot inputs (for example, required scopes and auto-execute configuration) to support review and incident response.

Executions SHOULD include an `executionId`, the associated `draftId`, a terminal status, and either a result object or an error string, plus timestamps for audit correlation.

C. Draft Lifecycle and State Machine

Drafts are long-lived governance objects and therefore require explicit lifecycle states. The reference runtime uses a small state machine aligned with operational workflows. Table III summarizes the semantics and transitions.

Figure 9 visualizes the same draft state machine.

Clients SHOULD treat `draft.status` as governance state, not as a substitute for an execution record. For asynchronous operation, clients MAY poll `GET /api/agent/v1/drafts/{id}` to retrieve the latest draft state and the most recent execution record.

D. High-Risk Safeguards: Preflight Hashing and Idempotency

For high-risk tools, OpenPort supports additional safeguards that make writes predictable under retries and reduce the chance of unintended side effects. These safeguards are used as gates for auto-execution and as evidence for operator review.

1) Preflight Impact Hash: The `POST /api/agent/v1/preflight` endpoint computes an impact summary and returns an impact hash. The preflight hash binds the action, payload, and impact summary using Eq. (2). Clients supply this value as `preflightHash` in subsequent `POST /api/agent/v1/actions` requests. When preflight is required for auto-execution, the server MUST deny auto-execution unless the client supplies a matching hash; denial codes include agent.preflight_required and agent.preflight_mismatch. When a deployment offers a preflight handle (`preflightId`) and the handle cannot be resolved, it SHOULD fail closed with agent.preflight_not_found rather than executing with a regenerated payload. This makes high-risk writes robust to agent retries and "time-of-check/time-of-use" confusion: the execution request is tied to the reviewed impact summary, not to an implicit UI interpretation.

2) Idempotency and Replay: Idempotency converts an unsafe retry pattern into a deterministic map (Eq. (5)). If a request carries an idempotency key and an execution already exists for the same $(\texttt{appId}, \texttt{idempotencyKey})$, the server SHOULD return the prior execution outcome and MUST NOT re-execute side effects. The reference profile uses agent.idempotency_replay for this case, enabling clients to treat the response as a success path. If high-risk auto-execution is enabled but an idempotency key is missing, the server MUST fail closed into a draft and return agent.idempotency_required.

E. Time-Bounded Auto-Execute (Optional)

Auto-execution is an explicit capability that MAY be enabled for a limited time window and tool allowlist. Eligibility is defined by Eq. (4). If an agent requests execution but $\mathrm{AutoExecAllowed}(r)$ is false, the server MUST return a draft and MUST return a stable denial reason code. In the reference implementation, common denial codes include agent.auto_execute_disabled, agent.auto_execute_expired, and agent.auto_execute_denied; high-risk guard failures use agent.preflight_required,

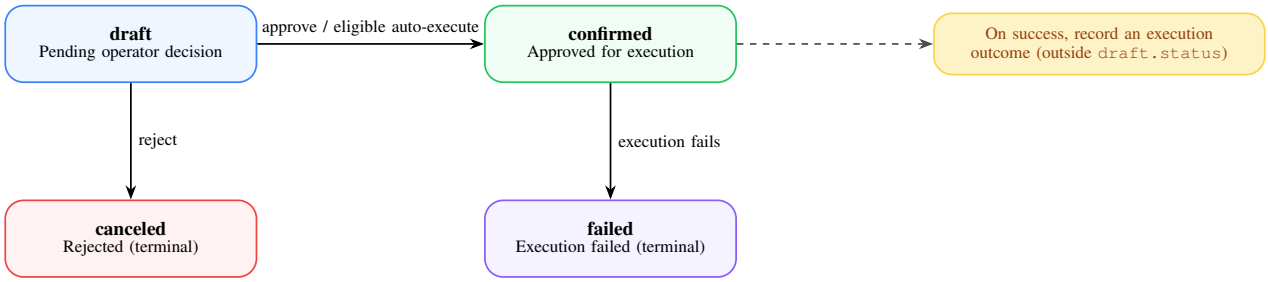

Fig. 9. Draft lifecycle state machine (reference profile).

agent.preflight_mismatch, and agent.idempotency_required. For high-risk auto-execution, the profile also requires a non-empty justification; missing justification is treated as invalid input and returns agent.action_invalid.

```
if execute && !forceDraft:
  if idempotencyKey && E(app,idempotencyKey) exists:
    return executed(replayed=true)
compute impact (+ preflight hash) for high-risk tools
denial = AutoExecAllowed(request) ? null : <reason code>
draft = saveDraft(status = denial ? "draft" : "confirmed", snapshot=policyInputs)
audit(draft, denial)
if denial: return draft(denial)
execution = executeDraft(draft)
return executed(execution)
```

Listing 5. Server-side decision procedure for `POST /actions` (conceptual).

F. Operator Approval and Separation of Duties

Draft approval and rejection are performed through an admin control plane (for example, `POST /api/agent-admin/v1/drafts/{id` and `/reject`). This separation of duties is intentional: the same credentials used for agent access MUST NOT be sufficient to approve high-risk writes. Approval triggers execution with full audit correlation (draftId, executionId, operator identity), and rejection transitions the draft to `canceled`. Attempts to act on missing or finalized drafts SHOULD return stable denial codes such as agent.draft_not_found and agent.draft_already_final.

VII. Rate Limits and Abuse Controls

Rate limiting is the operational "safety valve" for exposed agent tooling. Even with correct scopes and policies, a compromised key or a runaway agent loop can amplify cost and availability impact through excessive calls, aggressive polling, or repeated retries. OpenPort therefore treats admission control as part of the protocol's governance surface.

A. Objectives

OpenPort rate limiting and abuse controls target three production objectives:

- **Bound blast radius under leakage**: constrain the maximum request volume per credential and per network origin, so a stolen key cannot rapidly enumerate or exfiltrate data.
- **Prevent cost amplification**: protect expensive endpoints (for example, `/actions` and `/preflight`) from retry storms and polling loops.
- **Keep behavior predictable**: return stable, machine-actionable semantics so agent runtimes can implement deterministic recovery (Table II) and conformance tests can verify consistent 429 behavior (Table V).

B. Rate Limit Dimensions and Default Policy

At minimum, OpenPort implementations MUST enforce rate limits by the *(credential, client IP)* tuple. The credential dimension prevents a single integration key from overwhelming the system; the IP dimension reduces abuse from shared or spoofed credentials and provides a coarse network control. More granular quotas are OPTIONAL but recommended for real deployments:

- per integration app (`appId`) to support multiple keys and rotations;
- per tenant/workspace/resource to prevent cross-tenant "fan out" even under valid scopes;
- per tool/action to assign lower budgets to high-cost operations;
- per endpoint class (discovery, reads, writes, polling) so that `/manifest` and `/drafts` do not consume the same budget as `/transactions`.

OpenPort does not mandate a specific algorithm, but it requires predictable 429 behavior across these dimensions. Admission control MUST be evaluated *before* executing any tool logic and before allocating durable governance objects (drafts, executions). In particular, rate-limited requests MUST NOT create drafts or side effects. More generally, let $\mathcal{G}$ denote the set of governance objects created by the write pipeline (drafts and executions). For a request r, define $\Delta\mathcal{G}(r)$ as the newly allocated governance objects during handling of r. OpenPort's hard-denial posture is:

$$\Delta\mathcal{G}(r) = \emptyset \quad \text{whenever } \neg\text{Allow}(r), \tag{10}$$

which formalizes "no drafts on denials": authentication failures, scope/policy denials, and admission-control denials must not allocate drafts or executions.

C. Fixed-Window Admission Rule

One simple and testable limiter is a fixed-window admission rule. For a given key identifier k and client IP u, let $N_{k,u}(t)$ be the number of accepted requests in the interval $[t - W, t)$ where W is the window size. The limiter admits a request at time t iff:

$$N_{k,u}(t) < L, \tag{11}$$

where L is the maximum number of requests per window. Token-bucket-style mechanisms [7], sliding-window, and multi-dimensional quota systems are compatible, provided they preserve stable 429 semantics and can be audited.

D. 429 Semantics and Client Backoff

When a request is denied due to admission control, the server SHOULD return HTTP 429 with a stable code agent.rate_limited and a parseable error envelope. If available, servers SHOULD include `Retry-After` guidance [3], [4]; otherwise, clients MUST apply a conservative backoff strategy (for example, exponential backoff with jitter) rather than retrying immediately. Clients SHOULD also:

- reduce polling frequency for draft status and admin queues;
- cache `/manifest` and avoid re-discovery on every call;
- reuse `idempotencyKey` on write retries so a delayed retry cannot duplicate side effects (Eq. (5)).

Stable rate-limit behavior is a governance requirement because it determines whether agent runtimes converge to safe behavior under pressure, or amplify load by retrying unpredictably.

```
{ "ok": false, "code": "agent.rate_limited", "message": "Rate limit exceeded" }
```

Listing 6. Example 429 response envelope.

E. Additional Abuse Controls

Rate limits are necessary but insufficient. OpenPort pairs admission control with additional "abuse hardening" requirements:

- **Network policy allowlists**: optional IP allowlists deny traffic outside expected network boundaries with a policy denial code.
- **Schema validation and bounded parsing**: invalid inputs MUST fail as 4xx errors with stable codes (for example, unknown action or invalid payload), and SHOULD NOT degrade into 5xx failures.
- **Emergency disablement**: operators SHOULD be able to disable an integration app quickly during incident response, cutting off traffic across all endpoints including discovery.

These controls are designed to be testable in black-box conformance harnesses and to integrate cleanly with audit trails.

F. Reference Profile Parameters

The reference runtime enforces rate limiting at the authentication layer, before executing any endpoint logic. This ensures uniform protection across `/api/agent/v1/*` endpoints, including discovery and preflight. The current reference profile applies Eq. (11) per {keyId, client IP} with $W = 60$ seconds and $L = 240$ requests per window, using a bucket key of the form `agent:{keyId}:{ip}`.

VIII. Auditability

Auditability is mandatory for safe agent tool exposure. Without an audit trail, operators cannot answer the core production questions: *what* was accessed, *what* was attempted, *which* credential was used, *which* policy decision was applied, and *what* side effects occurred. OpenPort therefore treats audit events as first-class protocol outputs: they are required for incident response, compliance, and reproducible debugging of agent behavior.

TABLE IV
MINIMUM AUDIT EVENT FIELDS (REFERENCE PROFILE).

Field	Meaning / usage
`id`, `created_at`	Server-generated event identifier and timestamp for ordering and retention.
`action`, `status`	Stable event name and outcome (`success`, `denied`, `failed`).
`code`	Stable reason code for denials (for example, agent.scope_denied) and for protection outcomes (for example, agent.rate_limited).
`app_id`, `key_id`	Integration identifiers for attribution, revocation correlation, and blast-radius analysis.
`actor_user_id`	Subject identity under which domain reads/writes are performed (service user or user).
`performed_by_user_id`	Approving operator identity for admin actions and write approvals (separation of duties).
`request_id`	Optional client correlation identifier to tie retries and multi-step sequences.
`draft_id`, `execution_id`	Write lifecycle correlation from intent (draft) to effect (execution).
`ip`, `user_agent`	Request metadata for abuse analytics, incident response, and allowlist debugging.
`details`	Bounded, redacted metadata (risk tier, tool name, preflight summary, redacted fields), excluding secrets.

A. Goals

OpenPort audit requirements target three concrete goals:

- **Accountability**: attribute requests and side effects to an integration app/key, an actor identity, and (when applicable) an approving operator.
- **Reconstructability**: enable an operator to reconstruct request intent, authorization decision, and write lifecycle from drafts to executions, including idempotent replays and preflight binding.
- **Operability**: provide stable machine-facing reason codes for denial paths so incidents can be triaged and automated recovery strategies can be implemented safely (Table II).

B. Event Taxonomy

OpenPort requires audit events for **allow**, **deny**, and **fail** paths. The protocol does not mandate an internal logging system, but it does require a stable event taxonomy that is independent of model runtime behavior. The reference profile uses dot-separated action identifiers to group events:

- **Agent data reads**: for example, agent.ledger.list, agent.transaction.list.
- **Agent write governance**: for example, agent.action.preflight, agent.action.draft.created, agent.action.auto_execute.requested, agent.action.execute, agent.action.idempotency_replay.
- **Admin control plane**: for example, agent_app.create, agent_key.create, agent_app.policy.update, agent_app.auto_execute.update, agent_key.revoke, agent_app.revoke, agent.draft.approve, agent.draft.reject.
- **Protective controls**: rate-limit denials and policy denials, recorded with stable reason codes such as agent.rate_limited, agent.token_invalid, agent.token_expired, agent.scope_denied, and agent.policy_denied.

Implementations MAY extend the action namespace, but they SHOULD preserve backward compatibility for core actions to support detection rules and long-lived operational dashboards.

C. Minimum Event Schema and Correlation

Audit events MUST be structured and machine-parseable. The reference runtime uses an event schema with the following minimum fields:

Correlation is a protocol design requirement: drafts and executions MUST be linkable to the audit trail. Fields MAY be null when they are not available (for example, invalid or missing credentials may prevent resolving `app_id` and `key_id`). In these cases, events SHOULD still capture request metadata and stable reason codes to support abuse detection and incident response. In particular:

- `requestId` (if supplied) SHOULD propagate into audit to correlate agent retries and preflight/action pairs.
- `draftId` MUST be recorded for draft creation, approval/rejection, and execution outcomes.
- `executionId` MUST be recorded for execution outcomes and for idempotent replays.

To make write outcomes reconstructable from audit alone, OpenPort also treats draft-to-execution linking as a protocol invariant. Let $\mathcal{D}$ be the set of drafts and $\mathcal{E}$ be the set of execution records. OpenPort requires:

$$\forall e \in \mathcal{E} : \exists! d \in \mathcal{D} \text{ s.t. } \mathsf{draftId}(e) = \mathsf{id}(d), \tag{12}$$

where $\exists!$ denotes existence and uniqueness, $\mathsf{draftId}(e)$ denotes the draft identifier carried by execution record e, and $\mathsf{id}(d)$ denotes the server-issued identifier of draft record d. In words: every execution record MUST reference exactly one recorded draft intent. This invariant ensures every execution can be traced back to a single policy snapshot and approval decision, and it gives clients and auditors a deterministic join key between intent and effects.

Operationally, Eq. (12) is a referential-integrity requirement on the write pipeline. The extraction function $\mathsf{draftId}(\cdot)$ reads the draft identifier field from an execution record (as emitted by the gateway), and $\mathsf{id}(\cdot)$ is the primary identifier of a persisted draft record. The uniqueness clause rules out ambiguous correlation under retries or duplicate drafts, and the existence clause rules out "side-effect" executions that bypass draft creation and approval entirely.

```
{
  "id": "aud_...",
  "created_at": "2026-02-15T00:00:00Z",
  "action": "agent.action.draft.created",
  "status": "denied",
  "code": "agent.auto_execute_disabled",
  "app_id": "app_...",
  "key_id": "key_...",
  "request_id": "req_...",
  "draft_id": "drf_...",
  "ip": "203.0.113.4",
  "details": { "actionType": "transaction.hard_delete", "risk": "high" }
}
```

Listing 7. Example audit event (conceptual).

D. Data Minimization and Redaction

Audit events are security artifacts and MUST minimize sensitive content. OpenPort requires:

- **No secrets in audit**: bearer tokens, refresh tokens, and raw credentials MUST NOT appear in `details` or free-form error strings.
- **Bounded details**: `details` SHOULD be size-bounded and SHOULD avoid full request payloads. For example, transaction exports should audit row counts and policy parameters, not raw exported data.
- **Redaction awareness**: when data policies redact fields (Eq. (9)), audit events SHOULD record redacted field paths and policy mode, without recording raw sensitive values.
- **Sanitized failures**: adapter- or domain-level execution failures SHOULD be mapped to safe error classes; raw stack traces and high-entropy payload fragments SHOULD NOT be recorded in audit sinks.

Operational logs MAY contain richer debugging context, but they SHOULD be separated from immutable audit sinks and protected with different access controls.

E. Integrity, Retention, and Access Control

Audit sinks SHOULD be append-only and durable (for example, WORM storage or an external SIEM pipeline) to support incident response and compliance. Implementations SHOULD support integrity protections (for example, event signing or hash chaining) to make tampering detectable, for example via transparency-log style, append-only Merkle tree constructions [8]. Access to audit streams MUST be restricted to operators and automated security systems; agent credentials MUST NOT be sufficient to read the audit stream. The reference profile exposes audit listing only through the admin plane (for example, `GET /api/agent-admin/v1/audit`) for demonstration; production deployments SHOULD export audit events to an external sink and SHOULD avoid making the entire audit stream queryable via a public-facing API.

IX. Normative Requirements and Verifiable Invariants

OpenPort expresses requirements using RFC 2119/8174 keywords (MUST/SHOULD/MAY) [9], [10]. In deployments, "conformance" is only meaningful if requirements are (i) verifiable from the outside and (ii) stable over time. OpenPort therefore defines a set of externally observable invariants and encourages implementations to publish which conformance profile(s) they satisfy.

A. Normative Language and Compliance Claims

Key normative requirements include:

- **MUST** enforce deny-by-default scope checks and server-side tenant boundaries.
- **MUST** support immediate revocation for keys and apps (including discovery).
- **MUST** default writes to drafts; auto-execute must be explicitly enabled and time-bounded.

- **MUST** return stable response envelopes and stable reason codes for denials and protective controls.
- **MUST** emit audit events for allow/deny/fail paths with stable reason codes.
- **MUST** implement operational rate limiting.
- **SHOULD** require preflight and idempotency for high-risk execution.

A deployment that claims conformance to an OpenPort profile MUST satisfy every requirement of that profile. Profiles are versioned: tightening semantics or expanding required behavior SHOULD create a new profile identifier rather than silently changing the meaning of an existing claim. Backward compatibility is defined in operational terms: stable envelopes, stable reason-code semantics, and stable discovery behavior under unchanged authorization inputs.

B. Conformance Profiles

OpenPort profiles are machine-readable descriptions of the protocol surface a server exposes. A minimal profile typically specifies: (i) required endpoints (method + path), (ii) response envelope requirements for success and error cases, and (iii) security minimums (for example, unauthenticated discovery is denied). The reference repository provides a minimal core profile and an executable runner that can validate a local reference runtime or a remote deployment.

```
{
  "requiredEndpoints": [
    { "method": "GET",  "path": "/api/agent/v1/manifest" },
    { "method": "POST", "path": "/api/agent/v1/actions" }
  ],
  "envelope": {
    "success": { "requiredFields": ["ok","code","data"] },
    "error":   { "requiredFields": ["ok","code","message"] }
          }
}
```

Listing 8. Conceptual conformance profile fragment.

Profiles SHOULD be conservative and domain-neutral: they should not assume any specific product schema, and they should avoid irreversible operations. Draft-first writes make conformance safe because write intents can be exercised without applying side effects. For production deployments, OpenPort SHOULD be validated via layered profiles: **core** (endpoint presence + envelope consistency), **authz** (scope/policy denials + revocation behavior), **writes** (draft-first and high-risk safeguards), and **abuse** (rate limits and non-5xx fuzz regression). The reference repository includes a minimal core runner; the remaining profiles are intended as extensions that gate high-stakes deployments.

C. Invariants as Executable Checks

This paper uses "invariant" in a strict sense: a statement about observable behavior that can be checked from the outside using only protocol interactions and stable reason codes. Stable reason codes are the shared interface between verification and operations: the same codes that drive client recovery (Table II) enable deterministic test oracles and predictable incident response.

Several OpenPort properties are amenable to lightweight formalization. The equations introduced earlier are not merely expository; they define executable properties:

- authorization-dependent discovery (Eq. (1))
- execution binding via preflight hash (Eq. (2))
- TOCTOU resistance via state witness preconditions (Eq. (3))
- replay safety via idempotency mapping (Eq. (5))
- deterministic allow/deny semantics and reason-code stability (Eq. (6), Eq. (7))
- bounded query windows and deterministic presentation (Eq. (8), Eq. (9))
- rate limiting as a testable admission rule (Eq. (11))
- explicit time-bounded auto-execute eligibility (Eq. (4))

Table V summarizes representative invariants and black-box checks.

These checks form a minimal "safety bar" for production exposure. Importantly, they are domain-neutral: they can be run against any OpenPort deployment regardless of its underlying application schema, because they rely on the gateway's governance semantics rather than product-specific logic.

D. From Invariants to Release Gates

In an open-source governance protocol, invariants act as regression budgets: changes are acceptable only if they preserve conformance and do not weaken security properties. The reference repository operationalizes this through a release-gate pipeline

TABLE V
REPRESENTATIVE INVARIANTS AND BLACK-BOX CHECKS.

Invariant (reference)	**Black-box check (observable)**
Envelope stability	All endpoints return the standard envelope; errors are parseable and include a stable `code`.
AuthZ-dependent discovery (Eq. (1))	With insufficient scopes/policy, `/manifest` omits restricted tools/fields; adding scopes changes the visible set without breaking schema.
Immediate revocation	After revocation, requests across endpoints (including `/manifest`) return 401 with a stable token denial code (for example, agent.token_invalid).
Allow/deny determinism (Eq. (6), Eq. (7))	For a given request shape, denial code matches the first failing predicate in the prescribed order; codes remain stable across versions.
Preflight binding (Eq. (2))	When preflight is required, missing hash yields a required denial; mismatched hash yields a mismatch denial; an unresolvable `preflightId` yields agent.preflight_not_found; matching hash permits progression to draft/execute.
State witness preconditions (Eq. (3); optional profile)	If a draft is bound to `stateWitnessHash`, execution revalidates current witness state; mismatch yields 409 with agent.precondition_failed and no side effects.
Idempotency (Eq. (5))	Two identical write intents with the same `idempotencyKey` return the same execution record and do not produce duplicate side effects.
Auto-execute gating (Eq. (4))	Execution is denied outside the configured window, without allowlist membership, or without required justification/guards; in these cases, a draft is returned with a denial code.
Rate limiting (Eq. (11))	Exceeding the configured limit yields 429 with a stable agent.rate_limited code; the request does not create drafts or side effects.
Audit completeness (Table IV)	Allow/deny/fail paths emit structured audit events with correlation identifiers; audit payloads do not contain secrets.

that combines build and unit tests, conformance checks, fuzz/abuse regression ("no 5xx" for malformed inputs), and safety scans to prevent accidental secrets or boundary leakage in public artifacts. This engineering discipline is part of the protocol's governance story: it makes the specification enforceable in practice and reduces the chance that future releases silently expand the agent's authority surface.

X. REFERENCE IMPLEMENTATION AND ENGINEERING METHODOLOGY

We provide a reference runtime to validate the specification and reduce adoption cost. The implementation is intentionally small and deterministic: it exists to make OpenPort's governance semantics executable, not to prescribe an application architecture. This section describes the reference boundary (what is standardized vs what remains behind adapters) and the engineering method used to ship the protocol safely as open source.

A. Reference Runtime Goals

The reference runtime is designed around four concrete goals:

- **Make governance semantics concrete**: implement the agent-facing endpoints, stable envelopes, and reason codes; enforce scopes and ABAC constraints; and implement the draft-first write pipeline (preflight hashing, idempotency, and optional time-bounded auto-execute).
- **Remain domain-neutral**: keep product schemas and business rules behind adapters so that adopting OpenPort does not require open-sourcing internal models.
- **Fail closed**: deny-by-default on discovery and execution; ensure denials do not allocate durable governance objects (drafts/executions) and do not execute domain side effects; return stable `agent.*` reason codes for safe client behavior.
- **Be externally verifiable**: ship conformance profiles and black-box tests that validate the invariants in Table V without requiring privileged database access.

The reference runtime uses in-memory implementations for the app/key store, audit sink, and rate limiter to keep the codebase minimal. Production deployments SHOULD replace these with durable stores and distributed admission control, while preserving the observable protocol semantics. The minimal runtime also intentionally omits some optional controls (for example, step-up verification and advanced quota dimensions); these can be introduced as additional profiles without changing the core protocol envelope.

B. Gateway Architecture and Component Decomposition

The reference runtime is organized as a gateway and a small admin plane. The HTTP layer performs schema validation and returns standardized envelopes; request handling is delegated to governance engines that implement authentication, authorization,

TABLE VI
REFERENCE RUNTIME COMPONENTS (ILLUSTRATIVE).

Component	Responsibility
HTTP gateway (`/api/agent/v1/*`)	Route registration, schema validation, and standardized success/error envelopes.
Auth + admission control	Bearer token parsing, app/key resolution, revocation and expiry checks, IP allowlists, and fixed-window rate limiting before any tool logic.
Policy engine	Scope checks and ABAC constraints (resource allowlists, bounded query windows, redaction), plus tenant/workspace boundary enforcement.
Tool registry	Authorization-dependent discovery output (`/manifest`) and tool bindings (metadata + input/output schemas + optional impact computation for high-risk tools).
Draft/execution store	Write intent tracking (drafts), execution results, and idempotent replay mapping by $(\texttt{appId}, \texttt{idempotencyKey})$.
Audit service	Structured allow/deny/fail events with correlation identifiers (app/key/actor/draft/execution); pluggable sinks.
Admin plane (`/api/agent-admin/v1/*`)	App/key lifecycle, policy and auto-execute updates, draft approval/rejection, and audit export.

write control, and audit emission. The module decomposition is designed to align directly with the invariants and reason codes described earlier. Table VI summarizes the main components and their responsibilities.

For simplicity, the reference runtime uses a deliberately minimal admin authentication mechanism. This is not a recommended security pattern: production deployments MUST protect the admin plane with operator authentication and authorization, and they SHOULD apply step-up for high-risk approvals.

C. Adapter Boundary and Tool Registry

OpenPort standardizes governance semantics, not domain semantics. The reference runtime therefore defines a narrow *domain adapter* boundary that exposes only the minimal reads and writes needed by tools, and it passes an `actorUserId` that represents the effective subject (service user or delegated user). This boundary is the main extraction guardrail: product-specific identity models, joins, and business rules remain private, while the gateway enforces uniform governance around them. In practice, adopting OpenPort is primarily an integration exercise: implement the adapter boundary for the target application and register the tool set to expose, while reusing the protocol-level enforcement for scopes, policies, drafts, audits, and rate limits.

Tools are registered in a tool registry that produces the authorization-dependent manifest (Eq. (1)) and resolves action implementations for `POST /api/agent/v1/actions`. Each tool carries:

- **governance metadata**: `requiredScopes`, risk tier, and whether confirmation is required;
- **machine-readable schemas**: input/output schemas for client-side validation and tracing;
- **execution binding hooks**: for high-risk tools, an optional impact computation used by `/preflight` and by Eq. (2).

This structure makes risk controls explicit: risk tier affects whether preflight binding and idempotency are required for execution, and it determines whether auto-execution can be enabled at all.

D. Conformance Kit and Executable Profiles

To keep the protocol testable across deployments, the repository includes a machine-readable conformance profile and an executable runner. A profile declares: (i) required endpoints (method + path), (ii) envelope requirements for success and error cases, and (iii) minimum security expectations (for example, unauthenticated discovery is denied). The runner can operate in two modes: an in-process mode against the reference runtime (HTTP injection), and a remote mode that validates a deployment using only HTTP requests and a provided agent token.

The current core profile focuses on safe, domain-neutral checks: it validates `/manifest`, a small read surface, preflight behavior for a high-risk tool, and draft retrieval. Because OpenPort defaults writes to drafts, conformance can exercise the write pipeline without requiring destructive side effects, which makes it feasible to run profiles in public CI. As deployments mature, OpenPort SHOULD be validated using layered profiles (core/authz/writes/abuse/admin) as described in Section IX.

E. Release Gates for Safe Open-Source Evolution

OpenPort is intended to be extracted from private systems without leaking secrets or private implementation details. The reference repository therefore treats release engineering as part of the governance story: protocol correctness is not sufficient if the open-source distribution can accidentally publish credentials, internal hostnames, or product markers.

The release gate pipeline operationalizes this constraint by combining:

- **Build and unit tests**: type checking and regression tests for security controls (scope/policy boundaries, redaction, draft approvals, immediate revocation).
- **Conformance profile execution**: a black-box runner that validates endpoint presence and baseline semantics.
- **Safety scans**: checks for accidental environment files, common secret patterns (keys, private key blocks, tokens), and private-environment markers.
- **Boundary leak scans**: project-specific keyword checks intended to catch accidental references to private codebases or infrastructure in public assets.

These gates reduce the risk that protocol iteration silently expands the agent authority surface or leaks sensitive material during open-source publication.

F. Fuzz/Abuse Regression and Negative Security Tests

Finally, the reference repository includes fuzz-style and abuse-oriented regression tests that target production failure modes rather than happy-path demos. The test suite checks that malformed-but-valid JSON inputs do not produce 5xx regressions, that OpenAPI contracts match the reference server routes, and that key security controls are enforced: cross-tenant boundary denials, IP allowlists, bounded query windows, draft approval flows, and unit-tested admission control logic (fixed-window limiting). Together with conformance profiles, these tests provide executable evidence that OpenPort's governance semantics remain stable across releases, which is a prerequisite for multi-model agent runtimes to implement safe client behavior.

XI. Validation and Preliminary Evaluation

OpenPort is a governance specification. We therefore evaluate it as a set of externally observable invariants rather than as a task-level benchmark ("success rate" or "fewer tokens"). The primary question is whether an agent runtime can converge to safe behavior under realistic production failure modes: insufficient scopes, policy denials, revocation, retries, prompt-injection-driven tool misuse, and overload.

A. Methodology: Artifact-Based Validation

We combine three validation artifacts that can be run without privileged access to a product database:

- **Black-box conformance**: a profile-driven harness that interacts only through protocol endpoints and validates invariants (Table V).
- **Negative security tests**: regression tests for cross-tenant boundaries, policy windows, redaction, key revocation, and approval flows.
- **Fuzz/abuse regression**: malformed inputs and abuse-oriented regressions to ensure predictable error behavior (no 5xx regressions; stable envelopes and machine-readable codes on 4xx paths).

This methodology mirrors real adoption: third-party integrators and agent runtimes observe only the gateway interface and reason codes, not the application's internal schema.

B. Experimental Setup

We run the validation suite against a reference deployment consisting of:

- an OpenPort gateway configured with a small, deterministic tool registry
- a domain adapter seeded with synthetic multi-tenant data (multiple ledgers/workspaces)
- multiple integration apps/keys to cover distinct authorization configurations:
  - a workspace-scoped app restricted to a single organization
  - a personal-scoped app with narrower resource allowlists
  - at least one key under rotation and one revoked key

Policies are configured to exercise ABAC constraints (allowed resource IDs, bounded query windows as in Eq. (8), and sensitive-field redaction as in Eq. (9)) and network constraints (IP allowlists).

C. Metrics

We report outcomes as protocol-level properties rather than task-level agent benchmarks:

- **Artifact pass/fail**: whether build, unit tests, conformance profiles, and safety scans pass at a pinned tag.
- **Invariant coverage**: which protocol invariants (Table V) are exercised by the validation artifacts at that tag.
- **Robustness budget**: whether malformed inputs and negative paths avoid 5xx regressions and preserve stable response envelopes.

Expanded profiles can additionally report protocol-level metrics such as reason-code stability (Eq. (7)), end-to-end 429 semantics (Eq. (11)), audit completeness (Table IV), and idempotent replay behavior (Eq. (5)).

TABLE VII
EVALUATION COVERAGE AT TAG v0.1.0 (REPRESENTATIVE).

Property / invariant	Artifact(s)	Status
Envelope stability	Core conformance runner; OpenAPI contract test	Covered
Unauthenticated discovery denied	Core conformance unauthorized /manifest	Covered
Policy boundaries (tenant/IP/max-days)	Security controls tests; policy tests	Covered
Draft-first pipeline + approval	App tests (draft + admin approve); conformance draft retrieval	Covered
Preflight hash produced	Conformance /preflight; app high-risk flow	Covered
State witness preconditions	Optional profile; execution-time revalidation regression planned	Future
Idempotency replay mapping	Specified; no dedicated replay regression yet	Future
Rate limiting end-to-end 429	Limiter unit test; extended profile recommended	Partial
Audit completeness (allow/deny/fail)	Schema and admin listing exist; coverage expansion planned	Future

D. Artifacts and Reproducibility

To keep the evaluation maintainable while the project evolves, we pin artifact-based claims to a tagged release. At v0.1.0, the reference repository includes a release gate script that executes build, unit tests, a core conformance profile in local mode, and safety/boundary-leak scans. The recommended entry point is:

```
npm run gate
```

Listing 9. Reproducible validation entry point (tag v0.1.0).

The conformance runner can also be executed independently (local or remote mode), allowing third-party deployments to publish a profile conformance claim without sharing internal schemas. Scripts and profile definitions may evolve; the v0.1.0 tag provides a stable reference point for the claims in this paper. Optional stronger profiles (for example, State Witness / Preconditions) are treated as extensions and are evaluated separately from the pinned v0.1.0 claims.

E. Coverage at Tag *v0.1.0*

Table VII summarizes how the v0.1.0 artifacts map to the protocol invariants. The intent is not to claim that all OpenPort requirements are fully validated by the minimal suite, but to make the validation surface explicit and externally reproducible. We label a property as *Covered* when a runnable artifact includes an automated regression that fails if the property is violated, *Partial* when validation exists only at the component level (for example, unit-tested admission control without an end-to-end 429 profile), and *Future* when the requirement is specified but not yet exercised by the minimal suite.

F. Preliminary Results (Tag *v0.1.0*)

At v0.1.0, the release gate passes end-to-end on Node ≥ 20:

- TypeScript build completes without errors.
- Unit tests pass (8 test files; 20 tests), covering policy boundaries, draft approval flows, immediate revocation, contract-to-route consistency, and fuzz regressions (80 malformed requests across action and query surfaces; no 5xx).
- The core conformance profile passes in local mode, validating the presence of 6 required endpoints, envelope stability, preflight behavior for a high-risk tool, and draft retrieval.
- The tag asserts 3 distinct stable reason codes in unit tests (agent.token_invalid, agent.policy_denied, agent.idempotency_required); expanding reason-code stability regressions is future work.
- Abuse-oriented regressions include a unit test of the fixed-window limiter; an end-to-end 429 "no drafts on denial" profile is future work.
- Safety and boundary-leak scans detect no accidental .env files, no common secret patterns, and no forbidden private markers in public assets.

These results do not claim production readiness of the minimal runtime; they demonstrate that the protocol's governance semantics are testable, executable, and regression-protected in an open-source workflow.

XII. RELATED WORK

OpenPort Protocol sits at the intersection of tool exposure for agent runtimes and production API security governance. Adjacent work typically provides either (i) machine-readable interface contracts and invocation mechanics, or (ii) delegated identity and token transport. Neither class typically specifies the closed-loop governance semantics required for safe agent tool exposure: deny-by-default discovery, least-privilege scopes plus ABAC constraints, risk-gated writes that fail closed into drafts,

stable reason codes, admission control that is side-effect free, and mandatory auditability. OpenPort makes these behaviors part of the protocol surface and pairs them with externally verifiable invariants (Section IX).

A. Tool Exposure and Schema Contracts

Interface contracts are frequently described using OpenAPI [11], which standardizes endpoint and schema descriptions. Agent tooling ecosystems similarly rely on machine-readable tool catalogs and input/output schemas to avoid brittle UI automation. These approaches address *what* tools exist and *how* to call them, but they often leave *when* a tool is visible, *how* denials are represented, and *how* writes are governed as implementation-defined. OpenPort is compatible with contract descriptions and tool registries, but it adds governance semantics that are critical under agent failure modes: authorization-dependent discovery (Eq. (1)), stable `agent.*` denial codes, and a draft-first write path that prevents side effects by default.

B. Agent Tool Exposure Protocols and Registries

Recent agent ecosystems standardize tool discovery and invocation by treating applications as "tool servers" and by exposing machine-readable tool metadata (names, descriptions, schemas) that a client runtime can enumerate and call. This direction reduces reliance on UI automation and improves interoperability across models and runtimes. Model Context Protocol (MCP), for example, defines a server that exposes tools and schemas and a client runtime that can discover and invoke them [12], [13]. In the browser, WebMCP proposes a Web API (`window.navigator.modelContext`) for exposing JavaScript-implemented tools within a loaded page, explicitly framing such pages as a Web binding of MCP-style tool servers [14], [15]. However, tool exposure protocols often leave authorization closure, write governance, and operable failure semantics to each server implementation: which tools are visible under which credentials, how retries are guided under denial codes, how high-risk writes are gated behind review, and which invariants are externally verifiable. OpenPort positions governance as the missing standardization layer: it can bind to existing tool catalogs, but it requires the closed-loop semantics that make tool exposure safe under agent failure modes (draft-first writes, stable reason codes, admission control without side effects, and mandatory auditability).

C. Delegated Authorization, Revocation, and Token Binding

Delegated authorization commonly relies on OAuth 2.0 [6] and bearer token usage [16], with optional token introspection [17] and revocation [18]. These standards define identity delegation and token lifecycle primitives, but they do not mandate the governance semantics needed for agent tooling: least-privilege scope mapping, policy windows, risk-gated writes, and audit completeness. OpenPort therefore treats delegated identity as an input to a server-side mapping that yields minimal scopes and policy constraints, and it requires that revocation is immediately effective across discovery and tool calls with stable denial semantics.

Token theft is a central production risk. Proof-of-possession variants such as mutual TLS bound access tokens [19] and DPoP [20] reduce replay by binding a token to a key. OpenPort's minimal runtime uses bearer tokens for simplicity, but the protocol is compatible with PoP-style bindings: PoP strengthens authentication, while OpenPort focuses on authorization closure and safe write semantics even when a token is valid-looking.

D. Policy and Data-Domain Restriction

Classical access-control models such as RBAC [21] and ABAC-style policy constraints [5] provide flexible mechanisms for restricting the data domain and presentation surface beyond coarse scopes. OpenPort uses policy to encode resource allowlists, bounded query windows (Eq. (8)), and deterministic redaction (Eq. (9)). Unlike generic policy frameworks, OpenPort couples these constraints to the tool exposure surface itself: discovery, reads, drafts, and executions are all governed by the same enforcement posture and stable denial taxonomy.

E. Operational API Governance: Admission Control and Auditable Recovery

Operational patterns such as admission control and predictable 429 semantics [3], [4] are well understood for conventional APIs, but agent runtimes amplify their importance because retries, polling, and partial failures are common. OpenPort specifies rate limiting as a protocol invariant: rate-limited requests must fail before allocating drafts or side effects, and they must return stable, machine-actionable codes that drive deterministic client behavior (Table II). Auditability is similarly first-class: structured allow/deny/fail events and stable reason codes enable incident response and make governance properties testable as regressions rather than as informal best practices.

F. Write Preconditions and TOCTOU

Many APIs mitigate lost updates and time-of-check to time-of-use (TOCTOU) hazards using preconditions and optimistic concurrency patterns (for example, `ETag`-style validators and conditional requests) [3]. These mechanisms are typically scoped to a resource representation and are applied at the time of mutation. OpenPort adapts the same idea to agent tool exposure: high-risk writes may be subject to delayed operator approval, so the relevant "world state" can change between preflight and execution. The optional State Witness / Preconditions profile therefore binds a write intent to a server-observed witness hash (Eq. (3)) and requires execute-time revalidation. On mismatch, the server fails closed with a stable agent.precondition_failed reason code so clients can rerun preflight and obtain renewed approval rather than retrying blindly.

G. Capability Attenuation

Capability-based approaches model authorization as transferable, attenuable tokens. Macaroons, for example, represent bearer-style capabilities with contextual caveats that can restrict delegation [22]. OpenPort can be viewed as complementary: it expresses attenuation via scopes and ABAC policy constraints at the gateway, adds a governed write pipeline (draft-first, preflight binding, idempotency), and requires an audit trail so that authorization decisions are operable in real deployments.

Across these areas, OpenPort does not replace interface description or delegation standards; it constrains their composition for agent runtimes by defining a uniform recovery and governance surface. This emphasis on stable denial semantics, risk-gated writes, and verifiable invariants distinguishes OpenPort from specifications that stop at tool schema exposure or token transport.

XIII. Limitations and Future Work

OpenPort is designed to be a domain-neutral governance interface. That choice implies deliberate constraints.

A. Limitations

- **Not a prompt-safety solution**: OpenPort does not prevent prompt injection or malicious instruction following; it constrains the *effects* of tool calls via least privilege, draft-first writes, and operator review.
- **Trusted gateway and admin plane**: the threat model assumes the operator controls the gateway and the admin control plane. OpenPort does not defend against a malicious administrator.
- **Domain correctness is adapter-defined**: the protocol enforces scopes, policies, and write governance, but it cannot guarantee that an adapter's business logic is correct or safe; adoption requires careful adapter review and internal defense-in-depth.
- **Minimal runtime is not production-hardened**: at `v0.1.0`, governance state (apps/keys/drafts/ executions) and audit storage are in-memory by default, and admission control uses a process-local fixed-window limiter. Multi-node deployments require durable storage for governance state, a distributed limiter, and consistent idempotency behavior across replicas.
- **Bearer tokens in the minimal runtime**: the reference implementation uses bearer tokens for simplicity; this does not provide proof-of-possession and therefore does not reduce replay risk under token theft.
- **Risk classification is operator-defined**: OpenPort enforces write safeguards based on tool risk metadata; incorrect risk labeling or incomplete impact computation can weaken protections and must be treated as a governance responsibility.
- **Audit integrity is out of scope for the minimal profile**: OpenPort mandates structured allow/deny/fail events with stable codes, but it does not require tamper-evident audit integrity (signing or hash chaining) in the minimal runtime; deployments must secure the audit sink and retention pipeline.
- **Optional controls are not fully realized**: some governance mechanisms (for example, step-up verification and richer per-tool quotas) are described as profile extensions but are not enforced by the minimal core profile.

B. Future Work

We view OpenPort profiles as the primary mechanism to evolve the protocol without breaking existing clients. Priority future work includes:

- **Standard delegated-auth bindings**: a hardened OAuth 2.0 binding profile that specifies claim-to-scope mapping, policy windows, and revocation behavior, including safe UX patterns for agent consent.
- **Proof-of-possession modes**: optional PoP bindings (for example, mTLS- or DPoP-style) to reduce replay under token theft, while preserving the same authorization and audit semantics.
- **Multi-node governance semantics**: a persistence profile and reference implementation for apps/keys/drafts/ executions (with migration guidance for systems that already have internal audit and authorization pipelines), plus guidance and conformance profiles for consistent idempotency mapping, draft/execution correlation, and rate limiting across replicas, including an end-to-end 429 profile that proves "no drafts on rate-limit denials."
- **Audit integrity and export**: standardized audit export formats and optional integrity protection (event signing or hash chaining) suitable for SIEM ingestion and compliance retention.
- **Expanded conformance profiles**: publish and validate layered profiles for authZ, writes, abuse controls, and admin-plane security; expand black-box tests for audit completeness and denial-path side-effect freedom.

- **Cost-aware quotas**: per-tool budgets and query-cost controls to bound expensive exports and large retrievals beyond fixed request rate limits.

XIV. Conclusion

Safe tool exposure for AI agents is primarily a governance problem: runtimes must be able to discover and invoke tools without expanding the authority surface, and operators must be able to revoke, rate-limit, and audit those capabilities under real-world failure and abuse. This paper introduced OpenPort Protocol, a governance-first specification for exposing application data and actions to AI agents through a stable, machine-readable tool interface. OpenPort closes the authorization loop by making tool discovery authorization-dependent, enforcing explicit scopes plus ABAC policy windows, and standardizing stable response envelopes and `agent.*` reason codes that enable deterministic client recovery. For writes, OpenPort defaults to draft creation and supports risk-gated execution via preflight impact hashing, idempotency, and separation-of-duties approvals, so that high-risk operations fail closed into reviewable intent rather than side effects. Operationally, admission control and auditability are treated as protocol invariants: rate-limited requests must be side-effect free, and allow/deny/fail paths must emit structured audit events for incident response and compliance.

A governance protocol is only as useful as its verifiability. By pairing a reference runtime with layered profiles, black-box conformance tests, and release-gate automation pinned to immutable tags (for example, `v0.1.0`), OpenPort makes safety properties executable and regression-protected as the standard evolves. OpenPort is intentionally model- and runtime-neutral and can bind to existing tool ecosystems while keeping governance server-side. Future work focuses on durable multi-node semantics for governance state, stronger token-binding options, and expanded conformance profiles that validate audit completeness and end-to-end abuse controls.

Stewardship

OpenPort Protocol is stewarded by **Accentrust** and the OpenPort Protocol authors: **Genliang Zhu**, **Chu Wang**, **Ziyuan Wang**, **Zhida Li**, and **Qiang Li**. The project is governed as an open specification paired with a reference runtime and executable conformance artifacts. Stewardship emphasizes: (i) security-first defaults over convenience-first defaults, (ii) stable, machine-actionable semantics (envelopes and reason codes) for long-lived agent runtimes, and (iii) verifiable profiles that prevent silent expansion of the agent authority surface.

Versioning and Compatibility

OpenPort uses explicit versioning in its URL path (for example, `agent/v1`). The reference runtime is released under Semantic Versioning with immutable annotated tags (for example, `v0.1.0`) used to pin evaluation claims (Section XI). Within a major version, backward compatibility means: stable response envelopes, stable reason-code semantics, and conservative additive evolution of tool metadata and endpoints. Changes that alter authorization semantics, reason-code meaning, or safety invariants SHOULD be introduced as a new profile identifier or a new major version rather than modifying existing conformance claims.

Profiles and Conformance Claims

Implementations SHOULD publish which OpenPort profiles they satisfy (core/authz/writes/abuse/admin). A conformance claim is meaningful only if it is executable: profiles should be machine-readable, runnable against a deployment in black-box mode, and stable over time. This paper treats conformance and release gates as part of the protocol: they are the mechanism that allows an open governance standard to evolve safely.

Specification Change Control

OpenPort is intended to be implemented by multiple agent runtimes and multiple applications; silent behavioral drift is therefore treated as a standards failure. Protocol changes that affect client recovery (response envelopes, reason codes, retry semantics) or write governance (draft lifecycle, approval/execution semantics) SHOULD be accompanied by conformance updates and negative tests that fail when invariants are violated. Breaking changes SHOULD be introduced via a major version bump or a new profile identifier rather than by changing the meaning of existing reason codes. Changes are proposed via public pull requests and are expected to include documentation updates and pass release gates (build, tests, conformance profiles, and safety scans) before being tagged as a stable release.

Open-Source Safety Boundary

OpenPort is intended to be extracted from private systems without leaking secrets, identifiers, or product-private logic. The reference repository operationalizes this boundary through release gates that scan for common secret patterns and private markers, and through a design rule that pushes domain-specific behavior behind adapter interfaces rather than embedding business schemas in core modules. This boundary is a stewardship requirement: open-source publication must not expand the authority surface or disclose private implementation details.

Security Disclosures

Because OpenPort is intended for production tool exposure, vulnerabilities and unsafe-by-default behaviors are treated as protocol issues. Security reports SHOULD be submitted privately to the maintainers rather than through public issue trackers, and should include impact, affected endpoints, and reproducible steps. Before `v1.0.0`, only the latest minor release is supported for fixes; issues are addressed by rolling forward via patch releases and immutable tags. Stewardship prioritizes rapid triage, reason-code and envelope stability for clients, and regression protection through release gates, negative security tests, and conformance profiles. Maintainers aim to acknowledge reports within three business days and provide a severity/scope decision within seven business days.

REFERENCES

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep 1975. [Online]. Available: https://doi.org/10.1109/PROC.1975.9939

[2] A. Rundgren, B. Jordan, D. Balfanz, and M. Nystrom, "Json canonicalization scheme (jcs)," RFC Editor, RFC 8785, 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8785

[3] R. Fielding, M. Nottingham, and J. Reschke, "Http semantics," RFC Editor, RFC 9110, 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[4] M. Nottingham and R. Fielding, "Additional http status codes," RFC Editor, RFC 6585, 2012. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6585

[5] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations," National Institute of Standards and Technology, NIST Special Publication 800-162, Jan 2014. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-162/final

[6] D. Hardt, "The oauth 2.0 authorization framework," RFC Editor, RFC 6749, 2012. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6749

[7] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, "A single rate three color marker," RFC Editor, RFC 2697, 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2697

[8] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," RFC Editor, RFC 6962, 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6962

[9] S. Bradner, "Key words for use in rfcs to indicate requirement levels," RFC Editor, RFC 2119, 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119

[10] B. Leiba, "Ambiguity of uppercase vs lowercase in rfc 2119 key words," RFC Editor, RFC 8174, 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8174

[11] OpenAPI Initiative, "Openapi specification," https://spec.openapis.org/oas/v3.2.0.html, Sep 2025, version 3.2.0. Accessed: 2026-02-16.

[12] Model Context Protocol, "What is the model context protocol (mcp)?" https://modelcontextprotocol.io/docs/getting-started/intro, Feb 2026, accessed: 2026-02-17.

[13] ——, "Model context protocol (mcp) specification repository," https://github.com/modelcontextprotocol/specification, Feb 2026, commit `90c550bf4a261f1a410d624686e8619403e4dbc4` (main). Accessed: 2026-02-17.

[14] B. Walderman, L. Lee, A. Nolan, D. Bokan, K. Sagar, and H. Van Opstal, "Webmcp: Enabling web apps to provide javascript-based tools," https://raw.githubusercontent.com/webmachinelearning/webmcp/971aa24aea2afd865ca8607ba79a486fc7429360/README.md, Aug 2025, first published: 2025-08-13. Accessed: 2026-02-17.

[15] B. Walderman, A. Nolan, D. Bokan, K. Sagar, and H. Van Opstal, "Webmcp api proposal," https://raw.githubusercontent.com/webmachinelearning/webmcp/971aa24aea2afd865ca8607ba79a486fc7429360/docs/proposal.md, Aug 2025, accessed: 2026-02-17.

[16] M. Jones and D. Hardt, "The oauth 2.0 authorization framework: Bearer token usage," RFC Editor, RFC 6750, 2012. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6750

[17] J. Richer, "Oauth 2.0 token introspection," RFC Editor, RFC 7662, 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7662

[18] T. Lodderstedt, S. Dronia, and M. Scurtescu, "Oauth 2.0 token revocation," RFC Editor, RFC 7009, 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7009

[19] B. Campbell, J. Bradley, and N. Sakimura, "Oauth 2.0 mutual-tls client authentication and certificate-bound access tokens," RFC Editor, RFC 8705, 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8705

[20] D. Fett, M. Jones, and J. Bradley, "Oauth 2.0 demonstrating proof-of-possession at the application layer (dpop)," RFC Editor, RFC 9449, 2023. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9449

[21] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, Feb 1996. [Online]. Available: https://doi.org/10.1109/2.485845

[22] A. Birgisson, J. G. Politz, U. Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/macaroons-cookies-contextual-caveats-decentralized-authorization-cloud/